



University of  
Pittsburgh

# Algorithms and Data Structures 2

## CS 1501



Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

# Announcements

- Upcoming Deadlines
  - Homework 6: this Friday @ 11:59 pm
  - Lab 5: next Monday @ 11:59 pm
  - Assignment 1 Late Deadline Wednesday Oct 12<sup>th</sup> @ 11:59 pm
- Autograder issues and debugging hints
- If you think you lost points in a lab assignment because of the autograder or because of a simple mistake
  - please reach out to Grader TA over Piazza
- **Student Support Hours** of the teaching team are posted on the Syllabus page

# Previous lecture

- Huffman Compression
  - How to compute character frequencies
- Run-length Encoding
- LZW
  - compression and expansion algorithms
  - implementation concerns
- Shannon's Entropy

# This Lecture

- Comparing LZW vs Huffman
- Burrows-Wheeler Compression Algorithm

# LZW implementation concerns: codebook

- How to represent/store during:
  - Compression
  - Expansion
- Considerations:
  - What operations are needed?
  - How many of these operations are going to be performed?
- Discuss

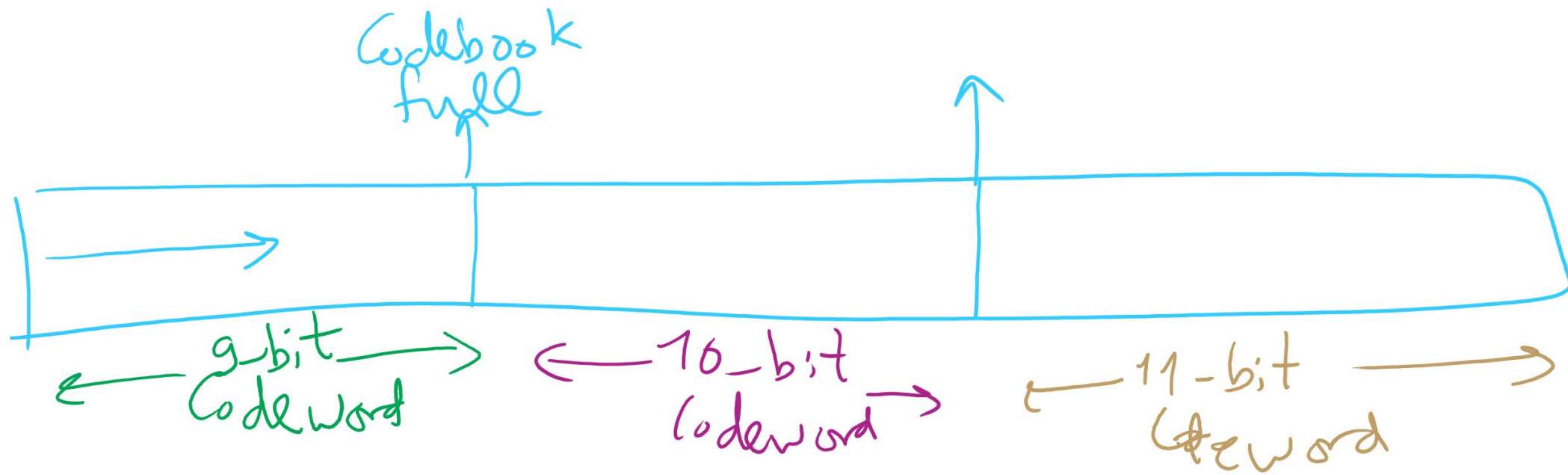
# Further implementation issues: codeword size

- How long should codewords be?
  - Use fewer bits:
    - Gives better compression earlier on
    - But, leaves fewer codewords available, which will hamper compression later on
  - Use more bits:
    - Delays actual compression until longer patterns are found due to large codeword size
    - More codewords available means that greater compression gains can be made later on in the process

# Variable width codewords

- This sounds eerily like variable length codewords...
  - Exactly what we set out to avoid!
- Here, we're talking about a different technique
- Example:
  - Start out using 9 bit codewords
  - When codeword 512 is inserted into the codebook, switch to outputting/grabbing 10 bit codewords
  - When codeword 1024 is inserted into the codebook, switch to outputting/grabbing 11 bit codewords...
  - Etc.

# Adaptive Codeword Size



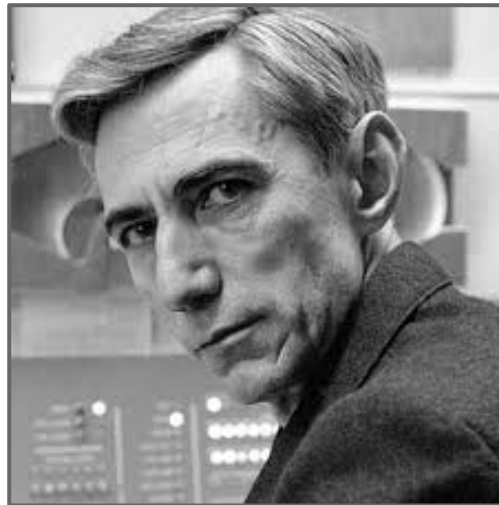


## Even further implementation issues: codebook size

- What happens when we run out of codewords?
  - Only  $2^n$  possible codewords for  $n$  bit codes
  - Even using variable width codewords, they can't grow arbitrarily large...
- Two primary options:
  - Stop adding new keywords, use the codebook as it stands
    - Maintains long already established patterns
    - But if the file changes, it will not be compressed as effectively
  - Throw out the codebook and start over from single characters
    - Allows new patterns to be compressed
    - Until new patterns are built up, though, compression will be minimal

# Can we reason about how much a file can be compressed?

- Yes! Using Shannon Entropy



# Information theory in a single slide...

- Founded by Claude Shannon in his paper "A Mathematical Theory of Communication"
- *Entropy* is a key measure in information theory
  - Slightly different from thermodynamic entropy
  - A measure of the unpredictability of information content
  - Example: which is more unpredictable?
    - a character that occurs with a probability of 0.5 or
    - a character that occurs with probability 0.25
    - which should have more entropy?

# Entropy

- Entropy equation:  $H(c) = -1 * \log_2 \text{Pr}(c)$ 
  - $\text{Pr}(c)$  is the probability of character  $c$
  - $\text{Pr}(c) = 0.5 \rightarrow \log_2(0.5) = -1 \rightarrow H(c) = 1 \text{ bit}$
  - $\text{Pr}(c) = 0.25 \rightarrow \log_2(0.25) = -2 \rightarrow H(c) = 2 \text{ bits}$
- On average, a lossless compression scheme cannot compress a message to have more than 1 bit of information per bit of compressed message
- By losslessly compressing data, we represent the same information in less space
  - Hence, 8 bits of uncompressed text has less entropy than 8 bits of compressed data

# Entropy of a file

- the entropy of a file is the average number of bits required to store a character in that file
- $H(\text{file}) = \sum_{\text{each unique character } c} H(c) * Pr(c)$
- How can we determine the probability of each character in the file?
  - depends on many factors
  - receiver and sender contexts
  - world knowledge
  - given none of that,  $Pr(c) = f(c) / \text{file size}$

# Entropy applied to language:

- Translating a language into binary, the entropy is the average number of bits required to store a letter of the language
- Entropy of a language \* length of message = amount of information contained in that message
- Uncompressed, English has between 0.6 and 1.3 bits of entropy per character of the message

# The showdown you've all been waiting for...

## HUFFMAN vs LZW

- In general, LZW will give better compression
  - Also better for compression archived directories of files
    - Why?
      - Very long patterns can be built up, leading to better compression
      - Different files don't "hurt" each other as they did in Huffman
        - Remember our thoughts on using static tries?

# So lossless compression apps use LZW?

- Well, gifs can use it
  - And pdfs
- Most dedicated compression applications use other algorithms:
  - DEFLATE (combination of LZ77 and Huffman)
    - Used by PKZIP and gzip
  - Burrows-Wheeler transforms
    - Used by bzip2
  - LZMA
    - Used by 7-zip
  - brotli
    - Introduced by Google in Sept. 2015
    - Based around a " ... combination of a modern variant of the LZ77 algorithm, Huffman coding[,] and 2nd order context modeling ... "



# DEFLATE et al achieve even better general compression?

- How much can they compress a file?
- Better question:
  - How much can a file be compressed by any algorithm?
- No algorithm can compress every bitstream
  - Assume we have such an algorithm
  - We can use to compress its own output!
  - And we could keep compressing its output until our compressed file is 0 bits!
  - Clearly this can't work
- Proofs in Proposition 5 of Section 5.5 of the text

# A final note on compression evaluation

- "Weissman scores" are a made-up metric for Silicon Valley (TV)



# Burrows-Wheeler Data Compression Algorithm

- Best compression algorithm for text
- The basis for UNIX's bzip2 tool

Adapted from: <https://www.cs.princeton.edu/courses/archive/spr03/cos226/assignments/burrows.html>

# BWT: Compression Algorithm

- Three steps
  - Cluster same letters as close to each other as well
    - Burrows-Wheeler Transform
  - Move-To-Front Encoding
    - Convert output of previous step into an integer file with large frequency differences
  - Huffman Compression
    - Compress the file of integers using Huffman

# BWT: Expansion Algorithm

- Apply the inverse of compression steps in reverse order
  - Huffman decoding
  - Move-To-Front decoding
  - Inverse Burrows-Wheeler Transform

# Move-To-Front Encoding

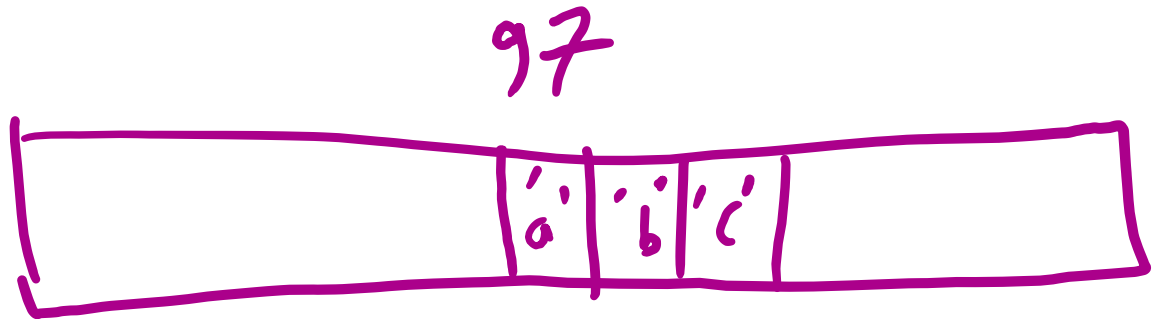
- Initialize an ordered list of the 256 ASCII characters
  - extended ASCII character  $i$  appears  $i$ th in the list
- For each character  $c$  from input
  - output the index in the list where  $c$  appears
  - move  $c$  to the front of the list

- Example:

**a** b b b a a b b b b a c c a b b a a a b c

97

- 'a' is 97 in ASCII



# Move-To-Front Encoding

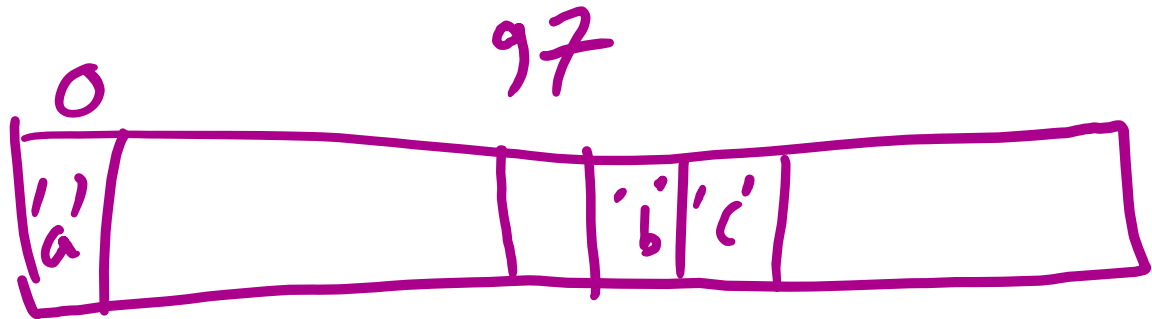
- Initialize an ordered list of the 256 ASCII characters
  - extended ASCII character  $i$  appears  $i$ th in the list
- For each character  $c$  from input
  - output the index in the list where  $c$  appears
  - move  $c$  to the front of the list

- Example:

a b b b a a b b b b a c c a b b a a a b c

97

- 'a' is 97 in ASCII



# Move-To-Front Encoding

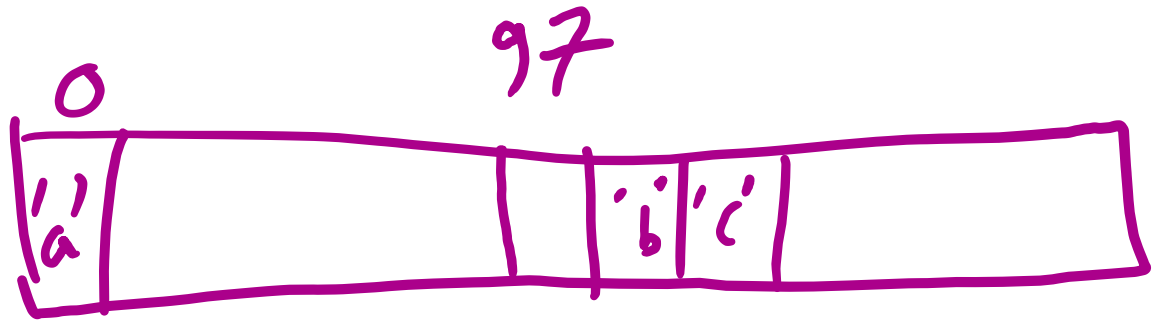
- Initialize an ordered list of the 256 ASCII characters
  - extended ASCII character  $i$  appears  $i$ th in the list
- For each character  $c$  from input
  - output the index in the list where  $c$  appears
  - move  $c$  to the front of the list

- Example:

a **b** b b a a b b b b a c c a b b a a a b c

97 98

- 'a' is 97 in ASCII





# Move-To-Front Encoding

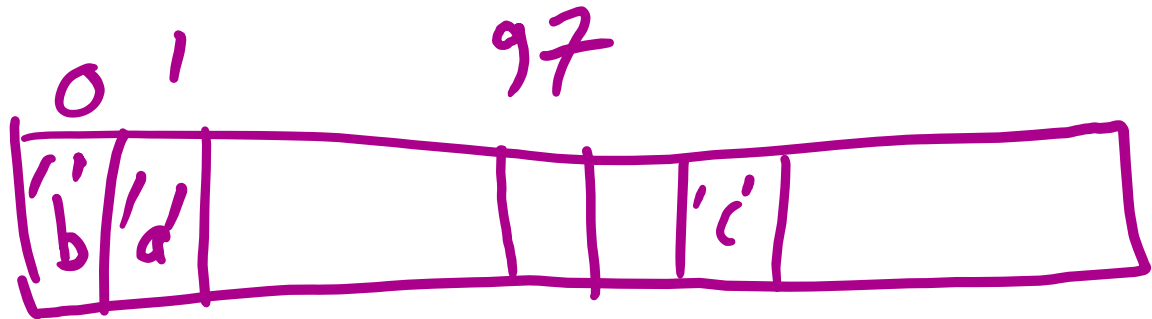
- Initialize an ordered list of the 256 ASCII characters
  - extended ASCII character  $i$  appears  $i$ th in the list
- For each character  $c$  from input
  - output the index in the list where  $c$  appears
  - move  $c$  to the front of the list

- Example:

a **b** b b a a b b b b a c c a b b a a a b c

97 98

- 'a' is 97 in ASCII



# Move-To-Front Encoding

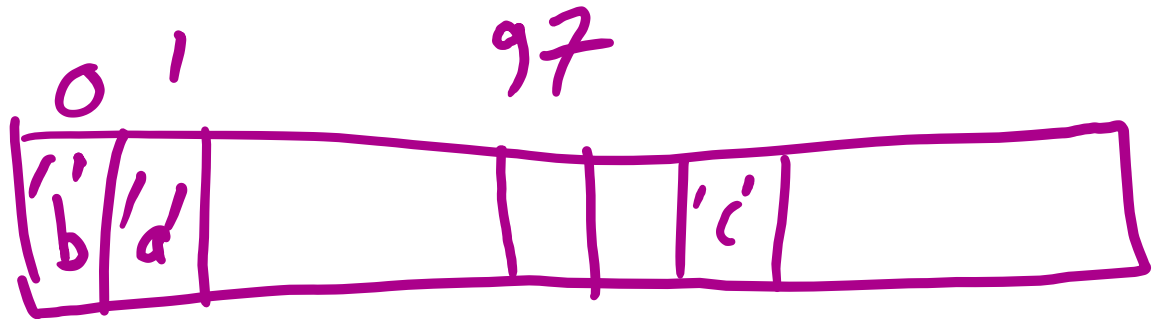
- Initialize an ordered list of the 256 ASCII characters
  - extended ASCII character  $i$  appears  $i$ th in the list
- For each character  $c$  from input
  - output the index in the list where  $c$  appears
  - move  $c$  to the front of the list

- Example:

a b **b** b a a b b b b a c c a b b a a a b c

97 98 0

- 'a' is 97 in ASCII



# Move-To-Front Encoding

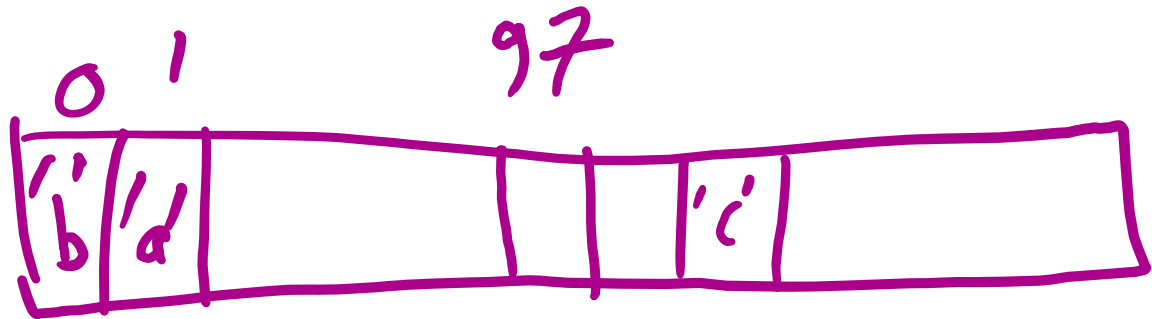
- Initialize an ordered list of the 256 ASCII characters
  - extended ASCII character  $i$  appears  $i$ th in the list
- For each character  $c$  from input
  - output the index in the list where  $c$  appears
  - move  $c$  to the front of the list

- Example:

a b b **b** a a b b b b a c c a b b a a a b c

97 98 0 0

- 'a' is 97 in ASCII



# Move-To-Front Encoding

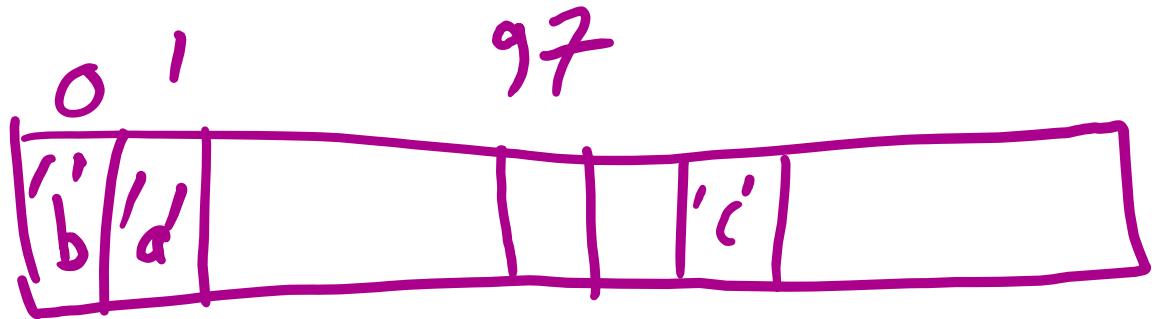
- Initialize an ordered list of the 256 ASCII characters
  - extended ASCII character  $i$  appears  $i$ th in the list
- For each character  $c$  from input
  - output the index in the list where  $c$  appears
  - move  $c$  to the front of the list

- Example:

a b b b **a** a b b b b a c c a b b a a a b c

97 98 0 0 1

- 'a' is 97 in ASCII



# Move-To-Front Encoding

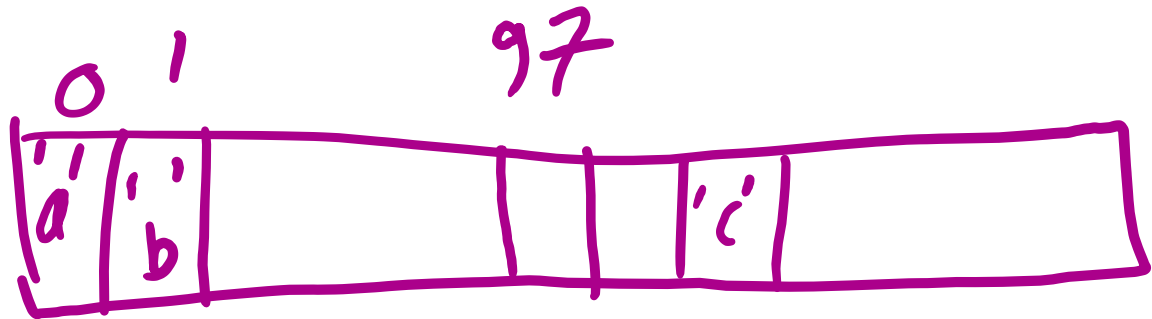
- Initialize an ordered list of the 256 ASCII characters
  - extended ASCII character  $i$  appears  $i$ th in the list
- For each character  $c$  from input
  - output the index in the list where  $c$  appears
  - move  $c$  to the front of the list

- Example:

a b b b **a** a b b b b a c c a b b a a a b c

97 98 0 0 1

- 'a' is 97 in ASCII



# Move-To-Front Encoding

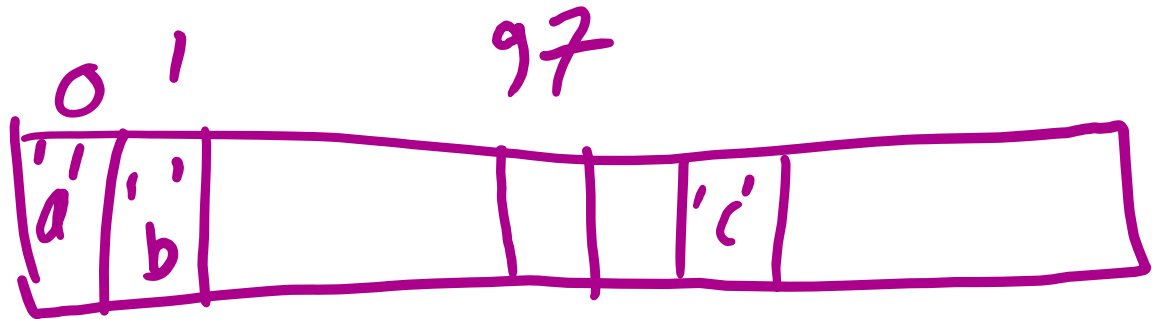
- Initialize an ordered list of the 256 ASCII characters
  - extended ASCII character  $i$  appears  $i$ th in the list
- For each character  $c$  from input
  - output the index in the list where  $c$  appears
  - move  $c$  to the front of the list

- Example:

a b b b a a b b b b a c c a b b a a a b c

97 98 0 0 1 0

- 'a' is 97 in ASCII



# Move-To-Front Encoding

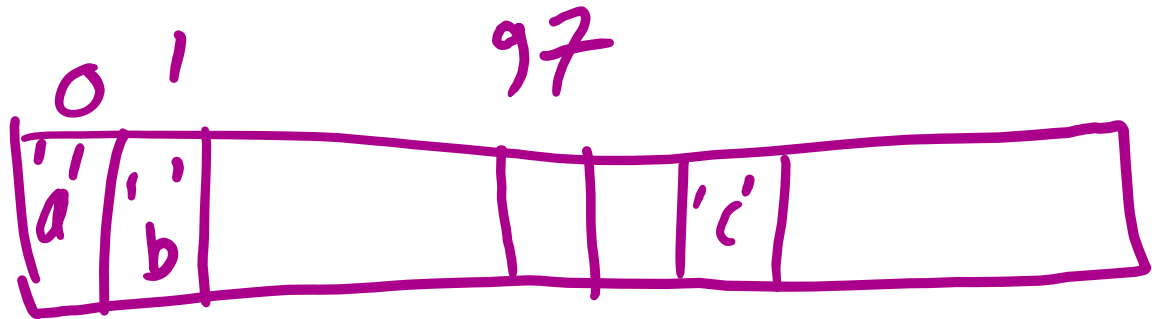
- Initialize an ordered list of the 256 ASCII characters
  - extended ASCII character  $i$  appears  $i$ th in the list
- For each character  $c$  from input
  - output the index in the list where  $c$  appears
  - move  $c$  to the front of the list

- Example:

a b b b a a **b** b b b a c c a b b a a a b c

97 98 0 0 1 0 1

- 'a' is 97 in ASCII



# Move-To-Front Encoding

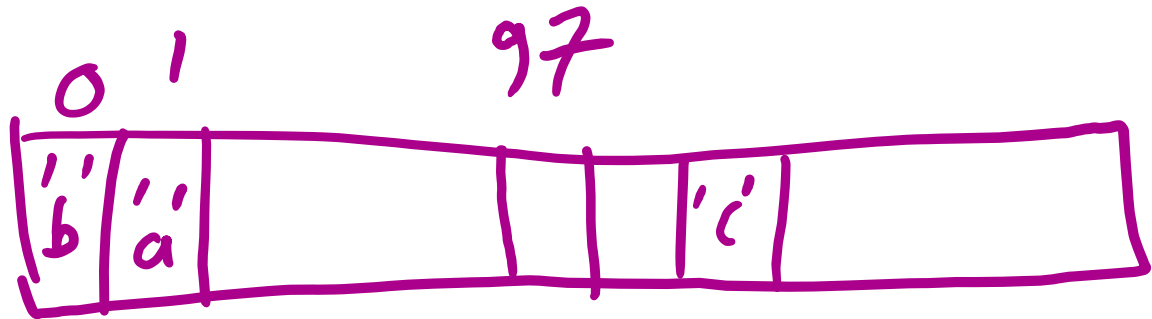
- Initialize an ordered list of the 256 ASCII characters
  - extended ASCII character  $i$  appears  $i$ th in the list
- For each character  $c$  from input
  - output the index in the list where  $c$  appears
  - move  $c$  to the front of the list

- Example:

a b b b a a **b** b b b a c c a b b a a a b c

97 98 0 0 1 0 1

- 'a' is 97 in ASCII





# Move-To-Front Encoding

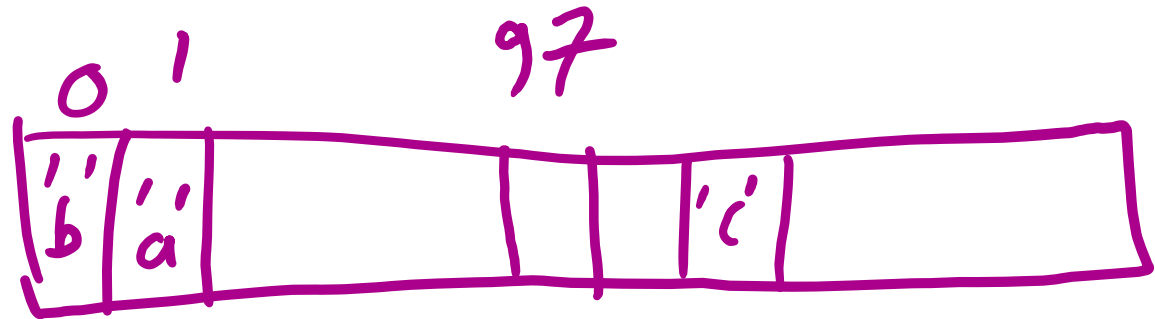
- Initialize an ordered list of the 256 ASCII characters
  - extended ASCII character  $i$  appears  $i$ th in the list
- For each character  $c$  from input
  - output the index in the list where  $c$  appears
  - move  $c$  to the front of the list

- Example:

a b b b a a b **b b b** a c c a b b a a a b c

97 98 0 0 1 0 1 0 0 0

- 'a' is 97 in ASCII



# Move-To-Front Encoding

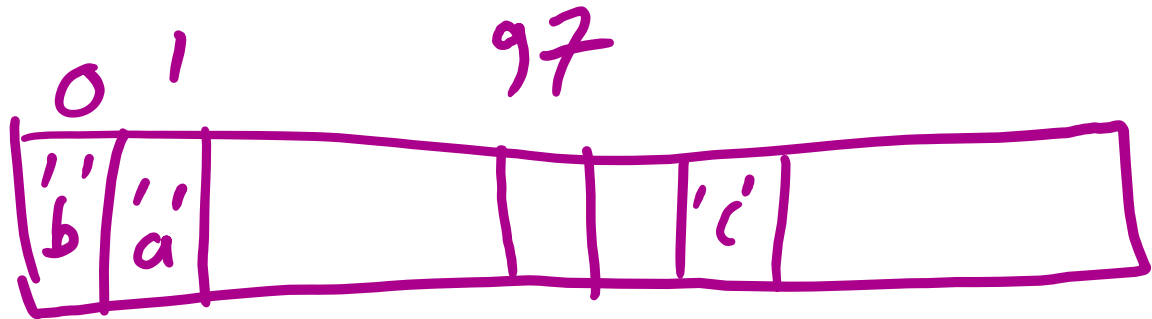
- Initialize an ordered list of the 256 ASCII characters
  - extended ASCII character  $i$  appears  $i$ th in the list
- For each character  $c$  from input
  - output the index in the list where  $c$  appears
  - move  $c$  to the front of the list

- Example:

a b b b a a b b b b **a** c c a b b a a a b c

97 98 0 0 1 0 1 0 0 0 1

- 'a' is 97 in ASCII



# Move-To-Front Encoding

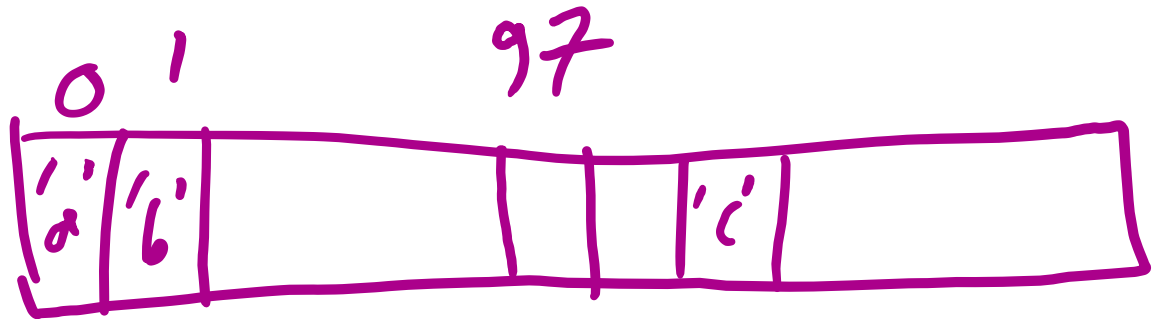
- Initialize an ordered list of the 256 ASCII characters
  - extended ASCII character  $i$  appears  $i$ th in the list
- For each character  $c$  from input
  - output the index in the list where  $c$  appears
  - move  $c$  to the front of the list

- Example:

a b b b a a b b b b a c c a b b a a a b c

97 98 0 0 1 0 1 0 0 0 1

- 'a' is 97 in ASCII



# Move-To-Front Encoding

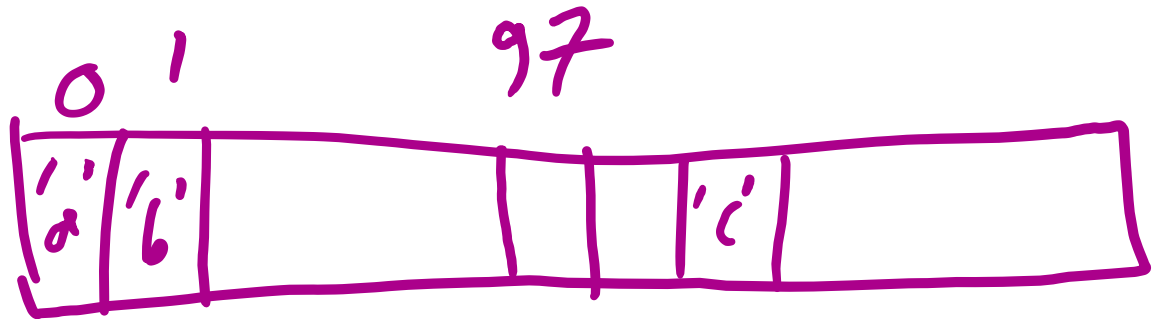
- Initialize an ordered list of the 256 ASCII characters
  - extended ASCII character  $i$  appears  $i$ th in the list
- For each character  $c$  from input
  - output the index in the list where  $c$  appears
  - move  $c$  to the front of the list

- Example:

a b b b a a b b b b a c c a b b a a a b c

97 98 0 0 1 0 1 0 0 0 1 99

- 'a' is 97 in ASCII



# Move-To-Front Encoding

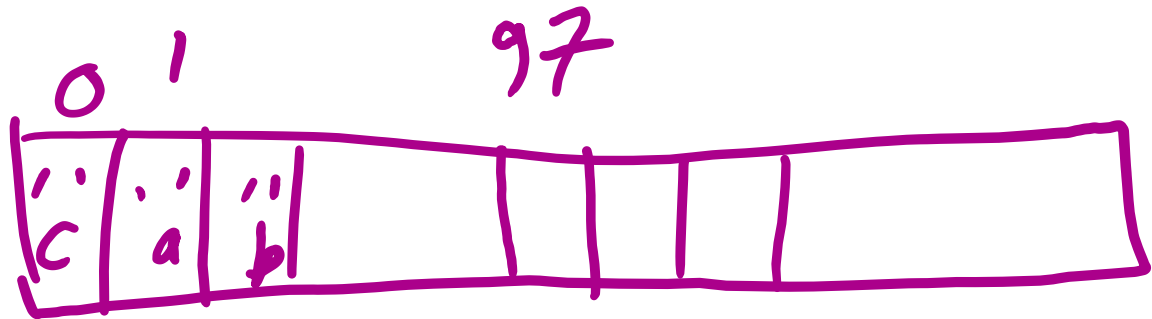
- Initialize an ordered list of the 256 ASCII characters
  - extended ASCII character  $i$  appears  $i$ th in the list
- For each character  $c$  from input
  - output the index in the list where  $c$  appears
  - move  $c$  to the front of the list

- Example:

a b b b a a b b b b a c c a b b a a a b c

97 98 0 0 1 0 1 0 0 0 1 99

- 'a' is 97 in ASCII



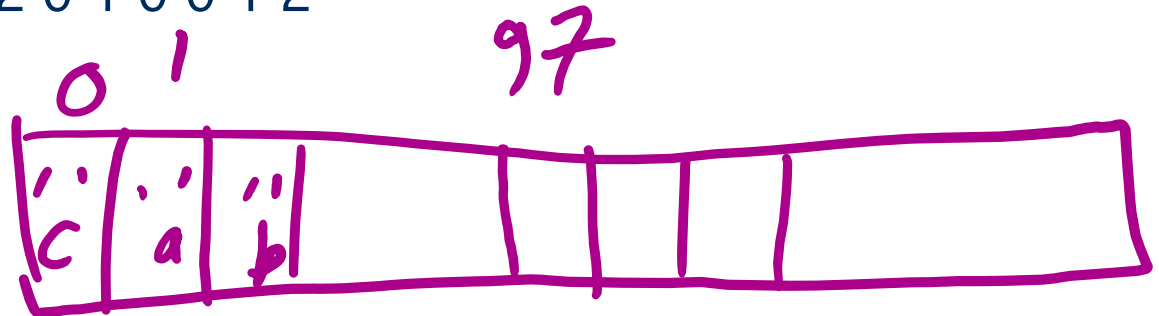
# Move-To-Front Encoding

- Initialize an ordered list of the 256 ASCII characters
  - extended ASCII character  $i$  appears  $i$ th in the list
- For each character  $c$  from input
  - output the index in the list where  $c$  appears
  - move  $c$  to the front of the list
- Example:

a b b b a a b b b b a c **c a b b a a a b c**

97 98 0 0 1 0 1 0 0 0 1 99 0 1 2 0 1 0 0 1 2

- 'a' is 97 in ASCII



# Move-To-Front Encoding

In the output of MTF Encoding, smaller integers have higher frequencies than larger integers

# Move-To-Front Decoding

- Initialize an ordered list of 256 characters
  - same as encoding
- For each integer  $i$  ( $i$  is between 0 and 255)
  - print the  $i$ th character in the list
  - move that character to the front of the list



# Burrows-Wheeler Transform

- Rearranges the characters in the input
  - lots of clusters with repeated characters
  - still possible to recover the original input
- Intuition: Consider **hen** in English text
  - most of the time the letter preceding it is t or w
  - group all such preceding letters together (mostly t's and some w's)

# Burrows-Wheeler Transform

- For each block of length  $N$ 
  - generate  **$N$  strings** by cycling the characters of the block one step at a time
  - sort the strings
  - output is the last column in the sorted table and the index of the original block in the sorted array

# Burrows-Wheeler Transform

- Example: Let's transform "ABRACADABRA"

- $N = 11$

- Cyclic Versions of the string:

- ABRACADABRA
- BRACADABRAA
- RACADABRAAB
- ACADABRAABR
- CADABRAABRA
- ADABRAABRAC
- DABRAABRACA
- ABRAABRACAD
- BRAABRACADA
- RAABRACADAB
- AABRACADABR

- After Sorting

- AABRACADABR
- ABRAABRACAD
- ABRACADABRA
- ACADABRAABR
- ADABRAABRAC
- BRAABRACADA
- CADABRAABRA
- DABRAABRACA
- RAABRACADAB
- RACADABRAAB



RDARCAAAABB

# Downsides of Burrows-Wheeler Algorithm

- Have to process blocks of input file
  - Compare to LZW, which processes the input one character at time
- The larger the block size, the better the compression
  - But, the longer the sorting time

# Repetitive Minimum Problem

- Input:
  - a (large) dynamic set of data items in the form of
- Output:
  - find a minimum item
- You are implementing an algorithm that repeats this problem
  - examples of such an algorithm?
    - Prim's, Huffman tree construction
- What we cover today applies to the repetitive maximum problem as well

# Let's create an ADT!

- The Priority Queue ADT
  - Primary operations of the PQ:
    - Insert
    - Find item with highest priority
      - e.g., findMin() or findMax()
    - Remove an item with highest priority
      - e.g., removeMin() or removeMax()
  - We mentioned priority queues in building Huffman tries
  - How do we implement these operations?
    - Simplest approach: arrays

# Unsorted array PQ

- Insert:
  - Add new item to the end of the array
  - $\Theta(1)$
- Find:
  - Search for the highest priority item (e.g., min or max)
  - $\Theta(n)$
- Remove:
  - Search for the highest priority item and delete
  - $\Theta(n)$
- Runtime for use in Huffman tree generation?

# Sorted array PQ

- Insert:
  - Add new item in appropriate sorted order
  - $\Theta(n)$
- Find:
  - Return the item at the end of the array
  - $\Theta(1)$
- Remove:
  - Return and delete the item at the end of the array
  - $\Theta(1)$
- Runtime for use in Huffman tree generation?



# Amortized Runtime

$$\text{Amortized runtime} = \frac{\text{Total runtime of a sequence of operations}}{\text{\#operations}}$$

# Amortized Time

$n$  inserts

$\Theta(n)$

$n^2$

$n$  Delete Min

$\Theta(1)$

$n = \Theta(n^2)$

+

$n^2$

Amortized Time =  $\frac{n^2}{\text{\#operations}}$

$= \frac{n^2}{2n} = \Theta(n)$


# So what other options do we have?

- What about a binary search tree?
  - Insert
    - Average case of  $\Theta(\lg n)$ , but worst case of  $\Theta(n)$
  - Find
    - Average case of  $\Theta(\lg n)$ , but worst case of  $\Theta(n)$
  - Remove
    - Average case of  $\Theta(\lg n)$ , but worst case of  $\Theta(n)$
- OK, so in the average case, all operations are  $\Theta(\lg n)$ 
  - No constant time operations
  - Worst case is  $\Theta(n)$  for all operations

# Is a BST overkill?

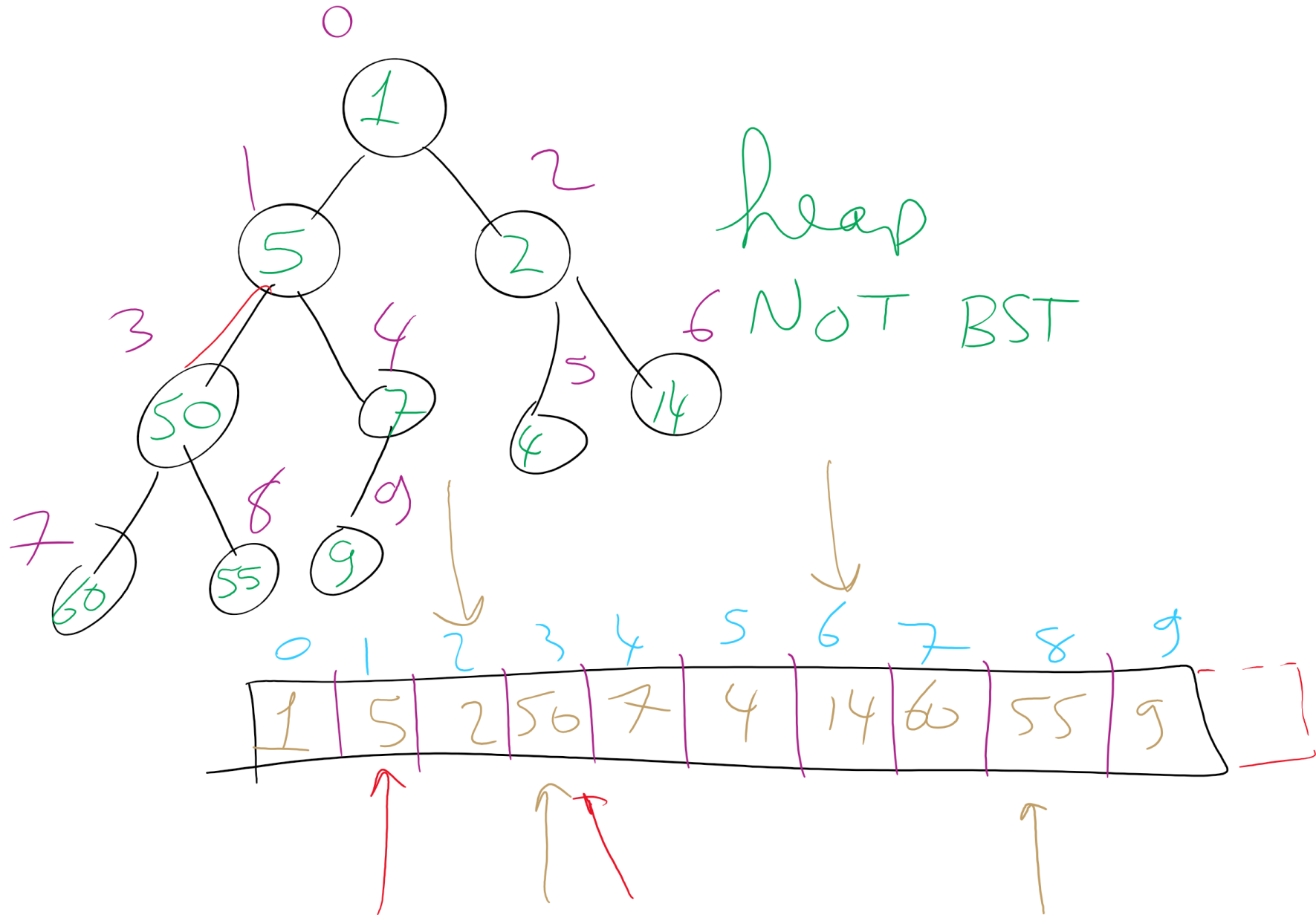
- Our find and remove operations only need the highest priority item, not to find/remove *any* item
  - Can we take advantage of this to improve our runtime?
    - Yes!

# The heap

- 
- A heap is complete binary tree such that for each node T in the tree:
    - T.item is of a higher priority than T.right\_child.item
    - T.item is of a higher priority than T.left\_child.item
  - It does not matter how T.left\_child.item relates to T.right\_child.item
    - This is a relaxation of the approach needed by a BST

*The heap property*

# Heap Example



# Heap PQ runtimes

- Find is easy
  - Simply the root of the tree
    - $\Theta(1)$
- Remove and insert are not quite so trivial
  - The tree is modified and the heap property must be maintained

# Heap insert

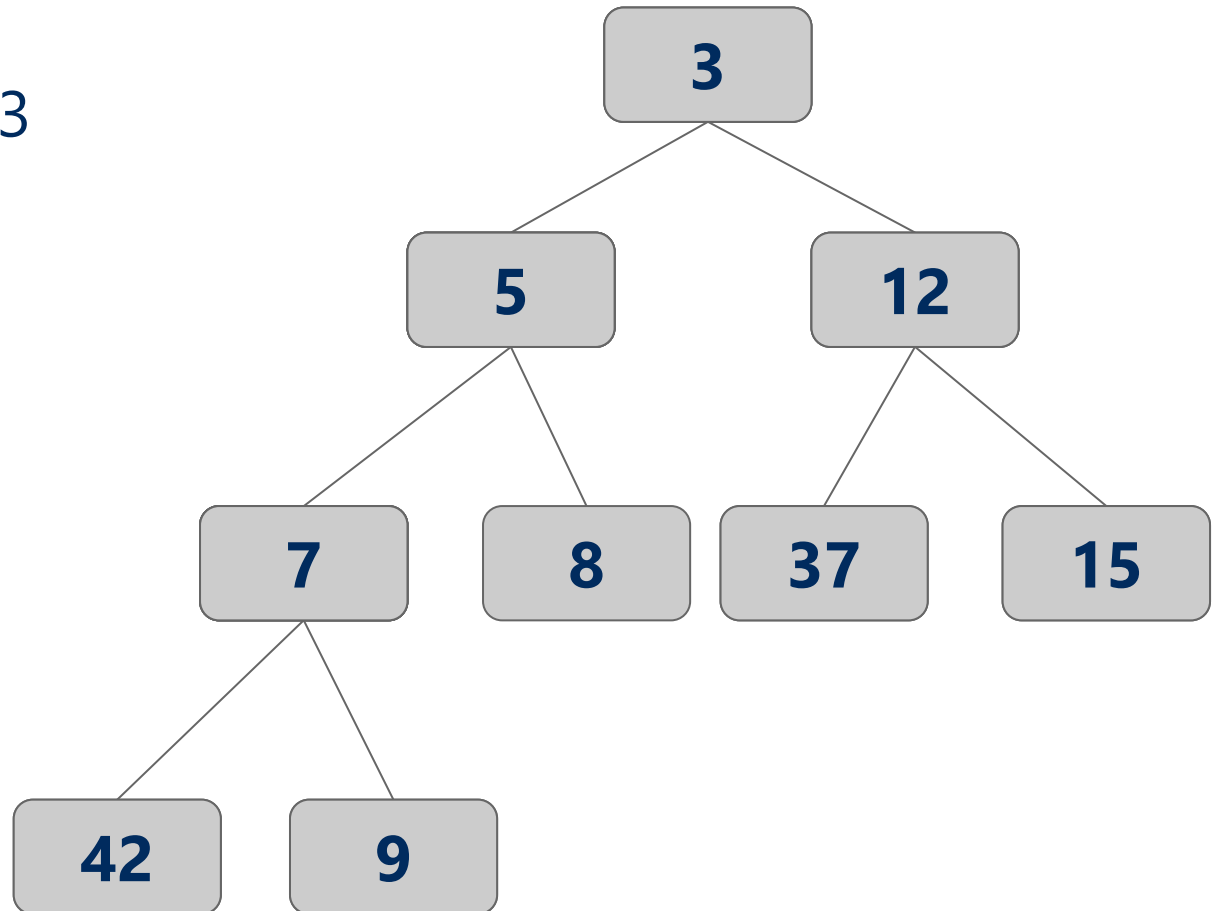
- Add a new node at the next available leaf
- Push the new node up the tree until it is supporting the heap property



# Min heap insert

Insert:

7, 42, 37, 5, 8, 15, 12, 9, 3



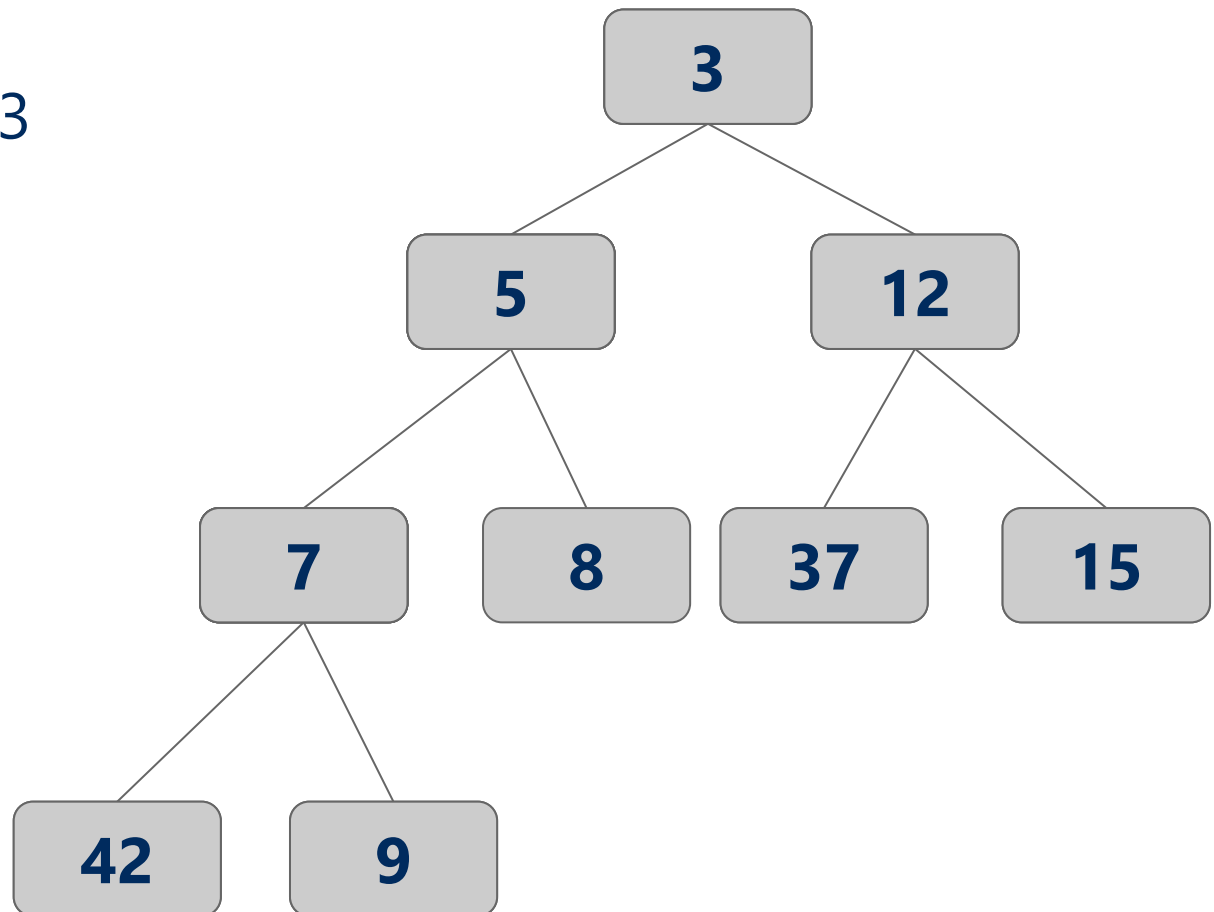
# Heap insert

- Add a new node at the next available leaf
- Push the new node up the tree until it is supporting the heap property

# Min heap insert

Insert:

7, 42, 37, 5, 8, 15, 12, 9, 3

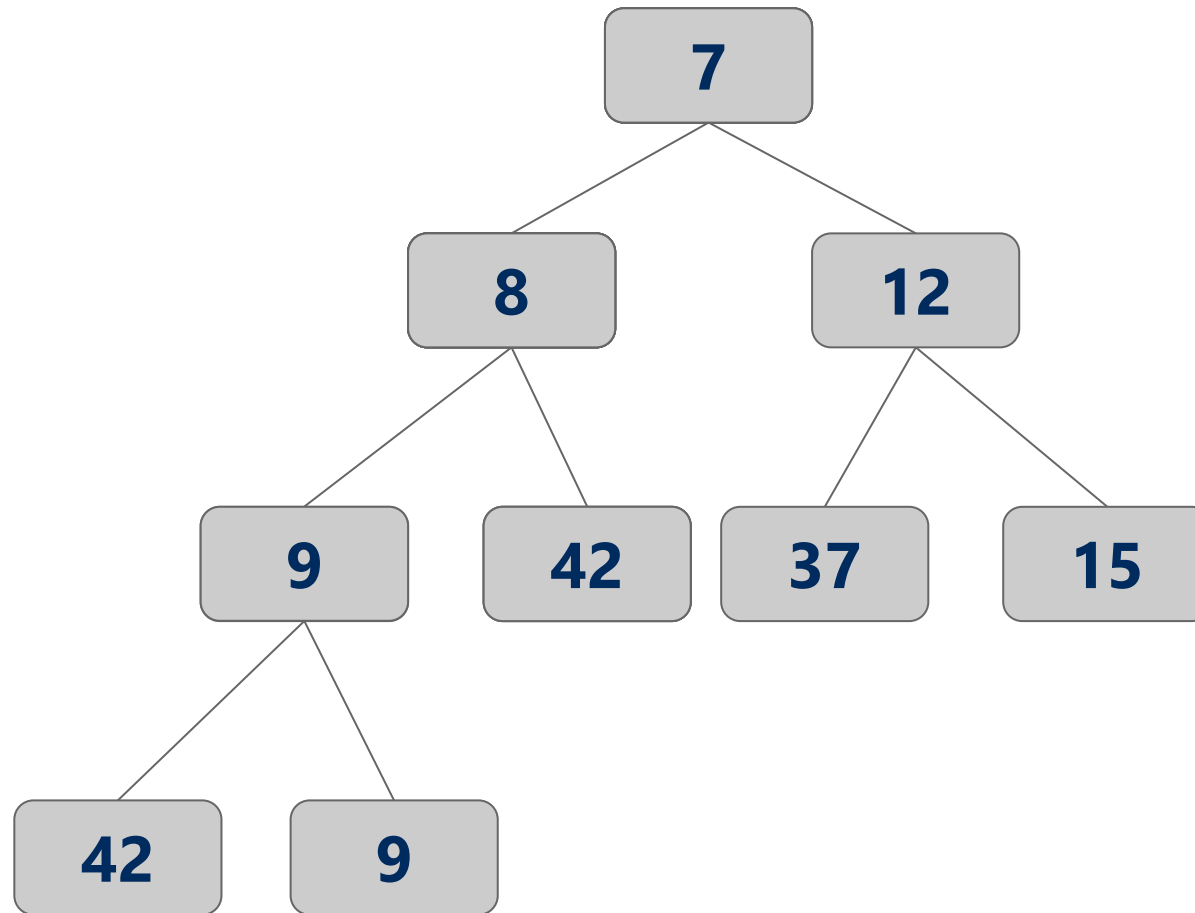


# Heap remove

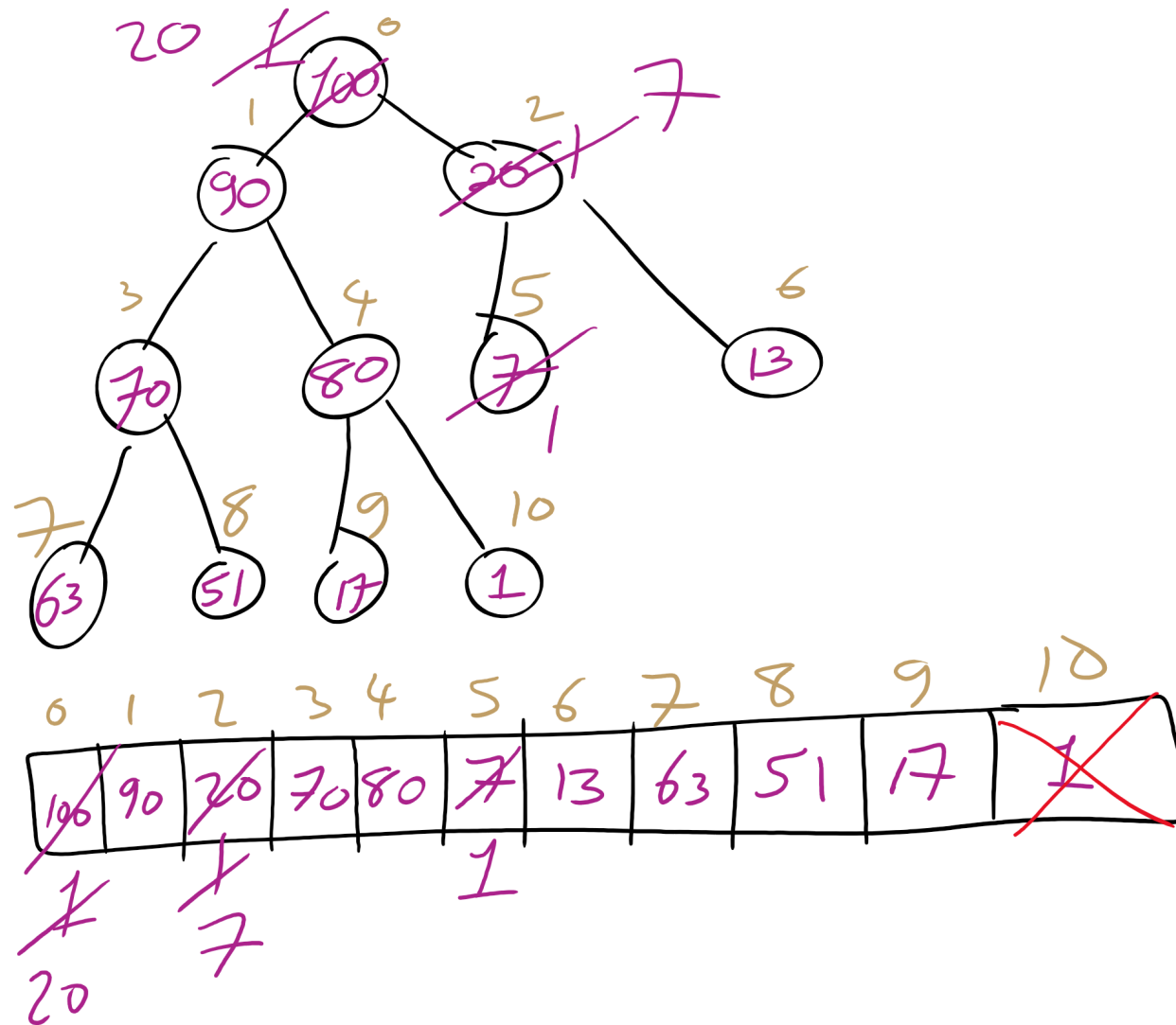
- Tricky to delete root...
  - So let's simply overwrite the root with the item from the last leaf and delete the last leaf
    - But then the root is violating the heap property...
      - So we push the root down the tree until it is supporting the heap property

# Min heap removal

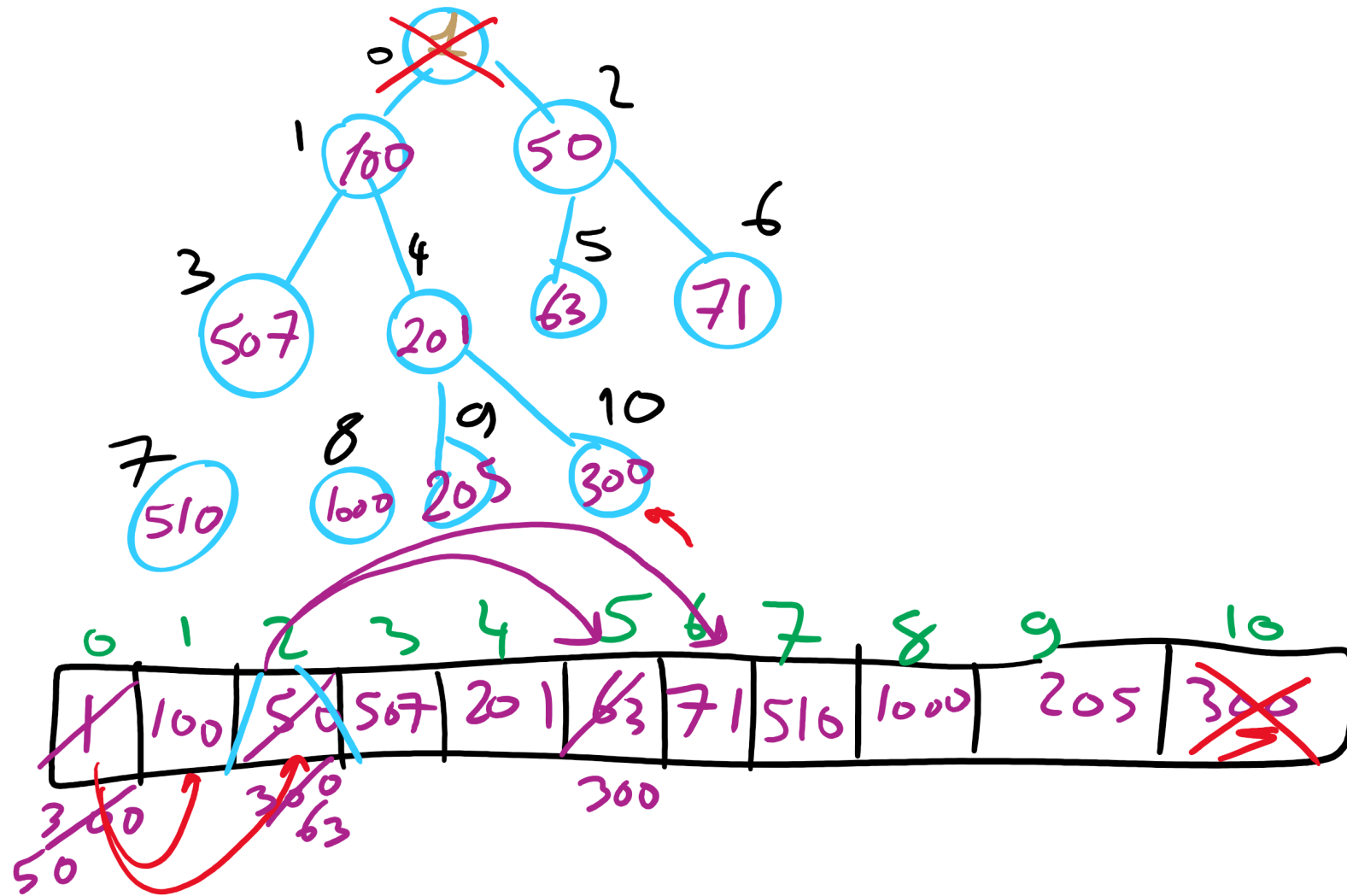
**NO!**



# Heap removeMax Example



# Heap removeMin Example



# Heap runtimes

- Find
  - $\Theta(1)$
- Insert and remove
  - Height of a complete binary tree is  $\lg n$
  - At most, upheap and downheap operations traverse the height of the tree
  - Hence, insert and remove are  $\Theta(\lg n)$



# Heap implementation

- Simply implement tree nodes like for BST
  - This requires overhead for dynamic node allocation
  - Also must follow chains of parent/child relations to traverse the tree
- Note that a heap will be a complete binary tree...
  - We can easily represent a complete binary tree using an array

# Storing a heap in an array

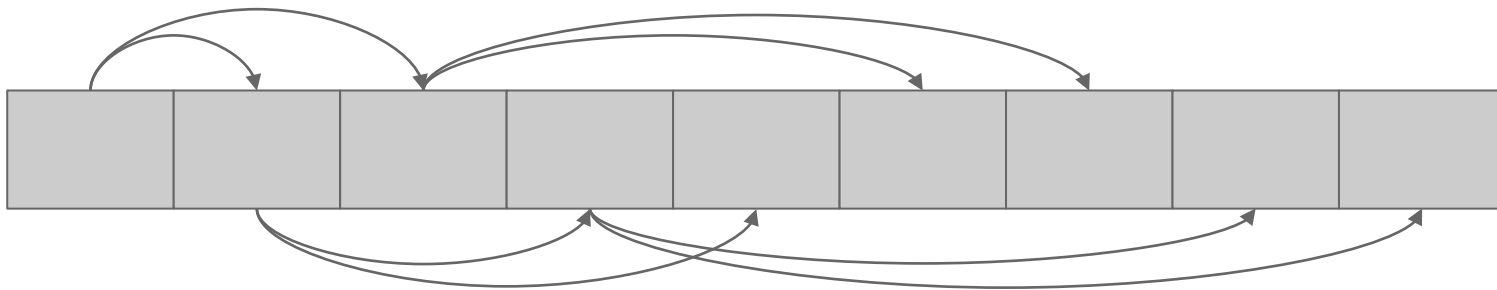
- Number nodes row-wise starting at 0
- Use these numbers as indices in the array
- Now, for node at index  $i$

- $\text{parent}(i) = \lfloor (i - 1) / 2 \rfloor$

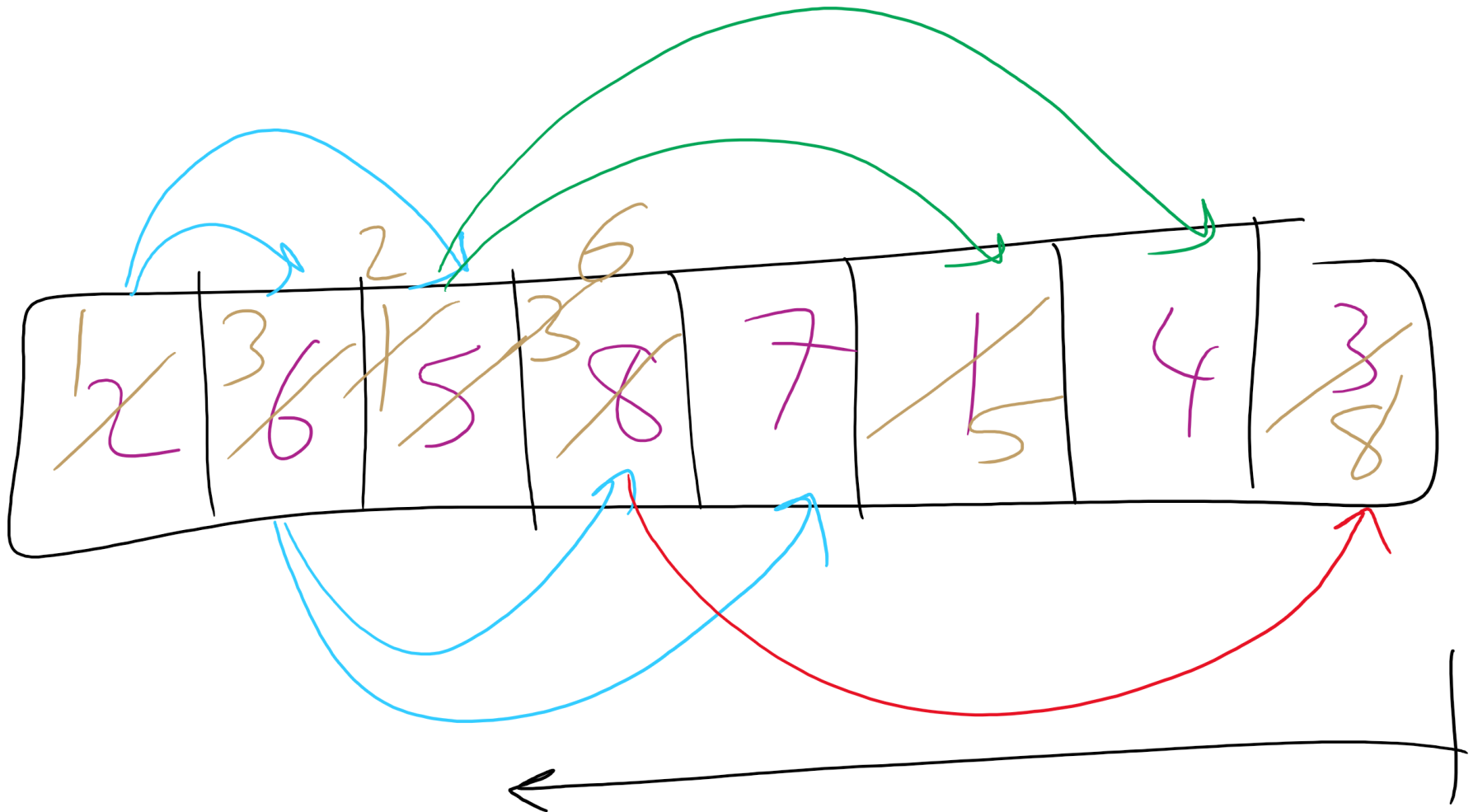
- $\text{left\_child}(i) = 2i + 1$

- $\text{right\_child}(i) = 2i + 2$

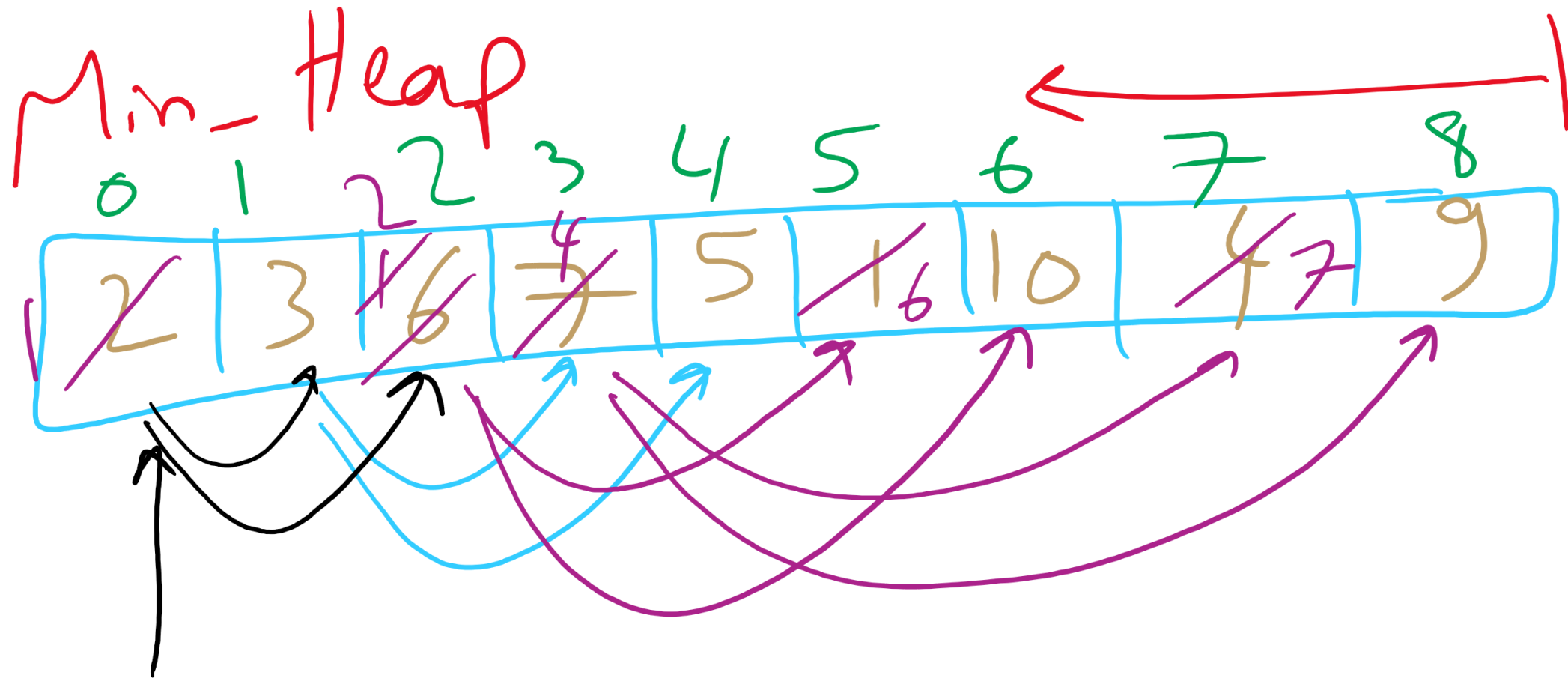
For arrays indexed from 0



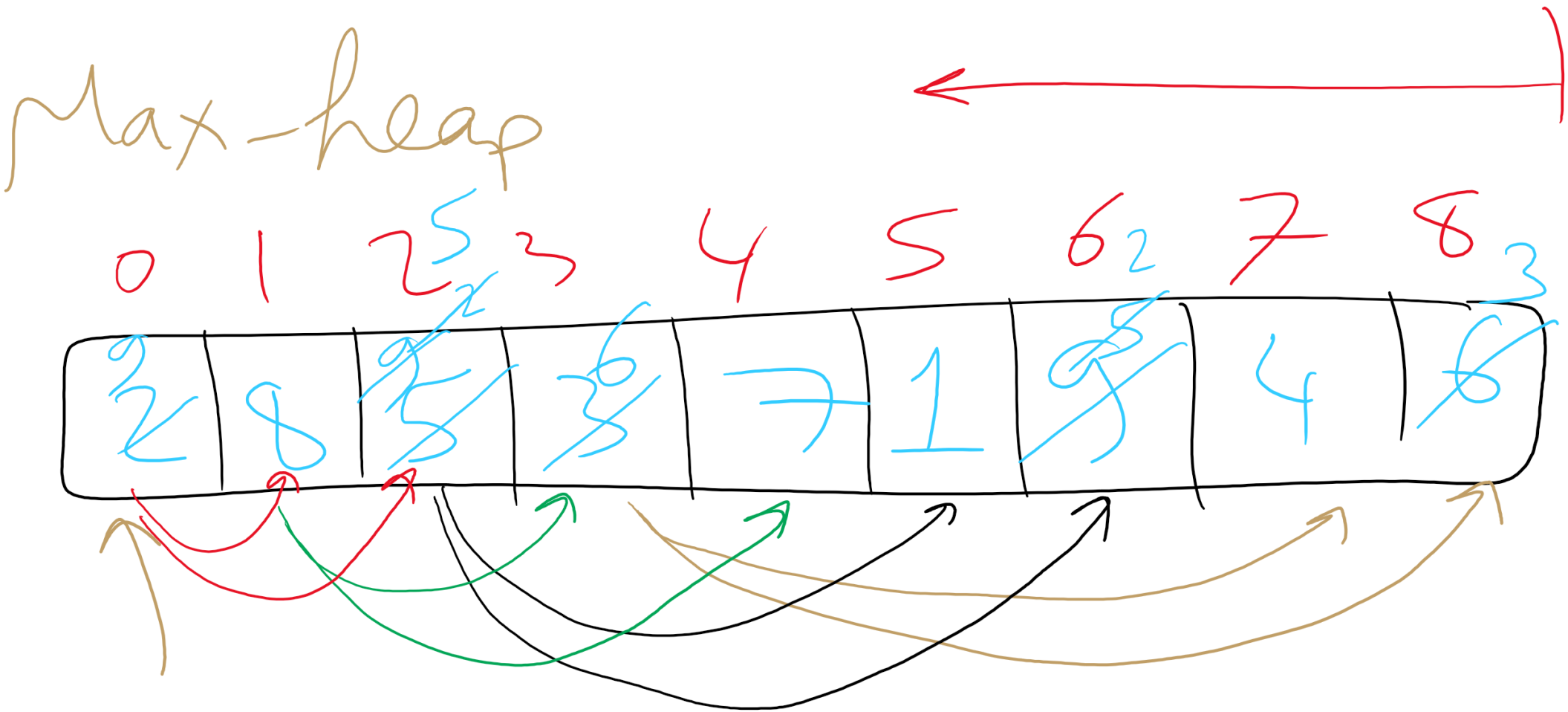
# Heapify Operation



# Heapify Example



# Heapify Example



# HeapSort Pseudo-code

HeapSort (Array  $a$ )

- Heapify( $a$ )

$\Theta(n)$

ascending  
MAX-HEAP

descending  
MIN-HEAP

end =  $n - 1$

for  $i = 0 \dots n - 1$

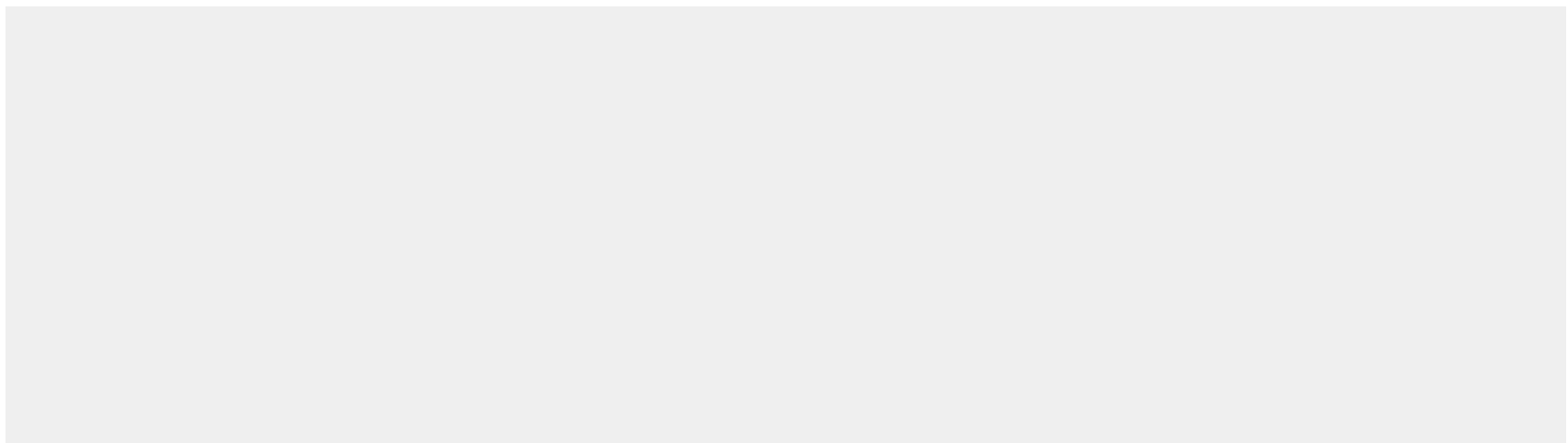
- remove highest priority  
item by swapping with  
the end of the array

- end --

$\log n$

# Heap Sort

- Heapify the numbers
  - MAX heap to sort ascending
  - MIN heap to sort descending
- "Remove" the root
  - Don't actually delete the leaf node
- Consider the heap to be from 0 .. length - 1
- Repeat



# Heap sort analysis

- Runtime:
  - Worst case:
    - $n \log n$
- In-place?
  - Yes
- Stable?
  - No



# Storing Objects in PQ

- What if we want to update an Object?
  - What is the runtime to find an arbitrary item in a heap?
    - $\Theta(n)$
    - Hence, updating an item in the heap is  $\Theta(n)$
  - Can we improve of this?
    - Back the PQ with something other than a heap?
    - Develop a clever workaround?

# Indirection

- Maintain a second data structure that maps item IDs to each item's current position in the heap
- This creates an *indexable* PQ

# Indirection example setup

- Let's say I'm shopping for a new video card and want to build a heap to help me keep track of the lowest price available from different stores.
- Keep objects of the following type in the heap:

```
class CardPrice implements Comparable<CardPrice>{  
    public String store;  
    public double price;  
    public CardPrice(String s, double p) { ... }  
    public int compareTo(CardPrice o) {  
        if (price < o.price) { return -1; }  
        else if (price > o.price) { return 1; }  
        else { return 0; }  
    }  
}
```

# Indirection example

- `n = new CardPrice("NE", 333.98);`
  - `a = new CardPrice("AMZN", 339.99);`
  - `x = new CardPrice("NCIX", 338.00);`
  - `b = new CardPrice("BB", 349.99);`
- 
- Update price for NE: 340.00
  - Update price for NCIX: 345.00
  - Update price for BB: 200.00

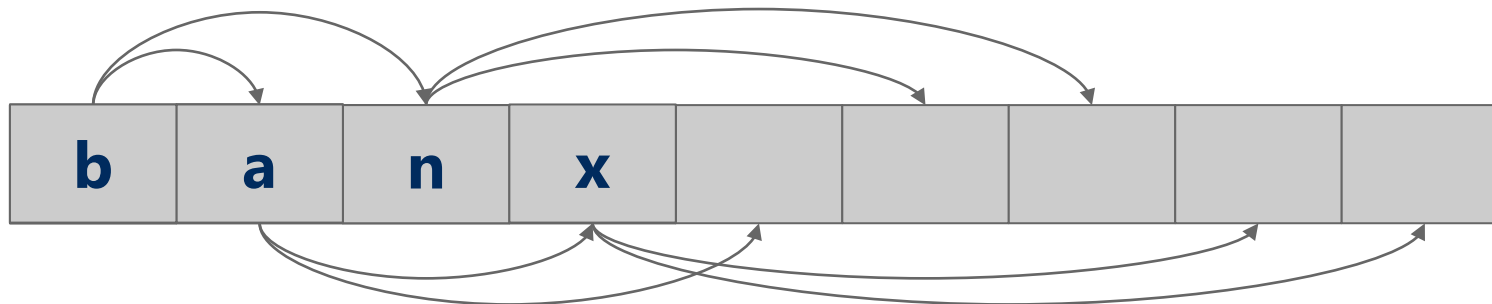
## Indirection

**"NE":2**

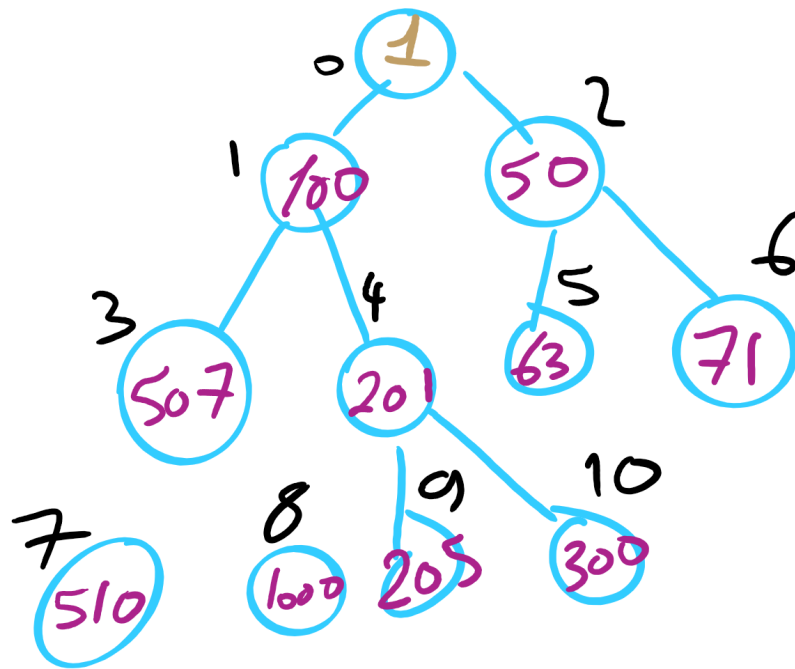
**"AMZN":1**

**"NCIX":3**

**"BB":0**



# Indexable PQ Example



1	0
50	2
63	5
71	6
-	
-	
-	
-	
-	
-	
-	

# Indexable PQ Example

